Modelling the Algebra of Weakest Preconditions

Abstract: Dijkstra's weakest precondition semantics, as presented in textbook form by Gries, may be viewed as an equational algebra. The problem then is to find a reasonable (set-theoretic) model of this algebra. This paper provides one.
Keywords: Guarded commands, Invariants, Predicate Transformers, Weakest preconditions.
Computing Review Categories: D.3.1, F.3.1, F.3.2.

1 Introduction

Programming language semantics comes in three flavours: operational, denotational and axiomatic. Of the latter, one of the best known is Dijkstra's *weakest precondition semantics*, introduced in Dijkstra [4] and [5], presented in textbook form in Gries [12], offered as a programming methodology in Backhouse [2], Dromey [7] and Morgan [15], and rejuvenated by Dijkstra and Scholten [6] as well as Holt [13].

The idea is that the meaning of a (nondeterministic) program α is entirely given by describing, for any given predicate Q which is desired to be true upon termination of α, the set of all those states s such that execution of α from s will result in α terminating in a state where Q is true. This set is called the *weakest precondition* of α with respect to Q, written $wp(\alpha, Q)$. By a *state* we mean the usual notion of a sequence of values of all program variables. The set of all states is here denoted by $\mathcal{S}$. A *predicate* is in the first instance an interpreted formula in a first-order language. However, it is common practise to equate a predicate Q with the set of all those states in which Q is true, and we adopt this useful ambiguity without further mention. Predicates therefore are subsets of $\mathcal{S}$. A program is *nondeterministic* if from a given input state different output states are possible. This holds in particular for atomic programs, so that if a program is viewed (operationally) as a sequence of atomic steps then we may think of an *execution tree* of states developing from any initial state.

Note that for any program α, '$wp(\alpha,—)$' is a mapping from predicates to predicates. Such a mapping is called a *predicate transformer* from the *power set* $\mathcal{P}(\mathcal{S})$ (*i.e.* the set of all subsets of $\mathcal{S}$) into itself. The Gries/Dijkstra methodology is to characterise programs by axiomatising the behaviour of predicate transformers. Since the axiomatisation is equational it is appropriate to think of it as an *algebra*, and it is this Gries/Dijkstra algebra to which our title refers.

The algebra of weakest preconditions has matured considerably from the original five 'healthiness conditions' in Dijkstra [5] to the self-contained algebraico-logical presentation of Dijkstra and Scholten [6]. For example, Dijkstra and Scholten ([6] p125) embrace a notion for which Dijkstra ([5] p76,77) could see no use: that of unbounded nondeterminism. (That is, from any given state there may be *infinitely many* possible next states for any program α.) We choose as representative the presentation of Gries [12], which, for the benefit of the reader, we quickly outline.

We will denote programs by $\alpha, \beta, \gamma, \ldots$, and predicates by $P, Q, R, \ldots$ (This notation differs from Gries.) There are certain atomic programs called *skip*, *abort* and (in Dijkstra and Scholten [6]) *havoc*. Another atomic program is the *assignment statement* '$z := e$', where z is a program variable and e is some expression (e.g. arithmetical). From the atomic programs compound programs are constructed in one of three ways. First, any two programs α and β mat be *composed* into another program $\alpha;\beta$, intuitively read as '*do* α, *then do* β'. Second, for any predicates $B_1, B_2, \ldots, B_n$ and programs $\alpha_1, \alpha_2, \ldots, \alpha_n$ there is an *alternative command* IF of the form:

$$if\ B_1 \rightarrow \alpha_1\ []\ B_2 \rightarrow \alpha_2\ []\ \ldots\ []\ B_n \rightarrow \alpha_n\ fi,$$

intuitively read as 'select some true B_i and execute the corresponding α_i'. [Note: the B_i's are called *guards* — which is why the Gries/Dijkstra language is sometimes referred to as a *guarded command language*.] Third, there is an *iterative command* DO of the form:

$$do\ B_1 \rightarrow \alpha_1\ []\ B_2 \rightarrow \alpha_2\ []\ \ldots\ []\ B_n \rightarrow \alpha_n\ od,$$

intuitively read as 'Repeat the following until no longer possible: select some true B_i and execute the corresponding α_i'.

The Gries/Dijkstra aim is to try and capture these intuitive meanings by conditions imposed on the '$wp(\alpha,—)$' predicate transformer. In Gries [12] these are the following: (We use the Gries numbering as an aid to the reader.)

The Algebra of Weakest Preconditions

(7.3) **Law of the Excluded Miracle:** $wp(\alpha, \emptyset) = \emptyset$

(7.4) **Distributivity of Conjunction:** $wp(\alpha, Q) \bigcap wp(\alpha, R) = wp(\alpha, Q \bigcap R)$

(7.5) **Law of Monotonicity:** If $Q \subseteq R$ then $wp(\alpha, Q) \subseteq wp(\alpha, R)$

(7.6) **Distributivity of Disjunction:** $wp(\alpha, Q) \bigcup wp(\alpha, R) \subseteq wp(\alpha, Q \bigcup R)$

(7.7) For any deterministic program α, $wp(\alpha, Q) \bigcup wp(\alpha, R) = wp(\alpha, Q \bigcup R)$

(8.1) $wp(skip, Q) = Q$

(8.2) $wp(abort, Q) = \emptyset$

(8.3) $wp(\alpha;\beta, Q) = wp(\alpha, wp(\beta, Q))$

(9.1.3) $wp(\text{`}z := e\text{'}, Q) = \{s|\ s[e/z] \in Q\}$

(10.3b) $wp(\text{IF}, Q) = (B_1 \cup B_2) \cap (\neg B_1 \cup wp(\alpha_1, Q)) \cap (\neg B_2 \cup wp(\alpha_2, Q))$

(11.2) $wp(DO, Q) = \bigcup_{n \geq 0} H_n(Q)$, where $H_0(Q) = \neg B \cap Q$, and

$H_{n+1}(Q) = H_0(Q) \cup wp(\text{IF}, H_n(Q))$ □

The reader familiar with Gries [12] will note that since our aim is to consider principles rather than details we have effected some simplifications, which we now explain. First, we have omitted multiple assignment (Gries [12] 9.4), since it merely repeats single assignment. Second, we have restricted IF to at most two guards (i.e. '$if\ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2\ fi$') since it is a mere notational matter to extend the results obtained for two guards to n guards. Third, we have restricted DO to *one* guard (i.e. '$do\ B \rightarrow \alpha\ od$', or more familiarly '$while\ B\ do\ \alpha$') since, as both Gries ([12] p139) and Dijkstra and Scholten ([6] p188) point out, in the presence of the general IF command the simple DO will suffice. Fourth, the notation '$s[e/z]$' is explained in §3.

As algebras go, one would expect the algebra of weakest preconditions to have some perspicuous *standard model*, preferably a set-theoretic one. (As, for example, a standard model for Boolean algebras is the calculus of sets.) But the matter is not that simple. The intuitive model of the algebra of weakest preconditions in Gries [12], for example, is an opportunistic hybrid of first-order logic, set theory and operational semantics, and it is not clear how this could give rise to a set-theoretic model. It seems that such a model must be reasonably sophisticated, in the sense of not just considering the input-output relation. Such an input-output approach has been explored for example in Sanderson [16], Blikle [3] and Schmidt and Ströhlein [18]. Programs are modelled as binary (input-output) relations over $\mathcal{S}$, and operations from the calculus of relations model operations on programs. This model has some pleasant features, but as is clearly pointed out in Gordon [9], it falters on an important point: (8.3) is not satisfied when composition of programs is interpreted as composition of relations.

Our aim in this paper is to model programs as *next-state relations*. On this approach a program run yields an *execution sequence* of states; because programs are nondeterministic any initial state s gives rise to an *execution tree*, and the meaning of a program is given by the set of all its possible execution sequences. Such a set will be called a *flowset* (by analogy

with *flow diagram*). In §2 we develop a calculus of flowsets, and this is used in §3 to verify all the Gries/Dijkstra conditions listed above. Thus the calculus of flowsets models the algebra of weakest preconditions. In §4 we briefly consider invariants (in an algebraic setting) in order to prove what Dijkstra [5] calls 'The Fundamental Invariance Theorem for Loops'.

We also have a secondary aim, which is pedagogical. Increasingly, Mathematics students are also doing Computer Science, and *vice versa*. To a mathematically mature student population weakest precondition semantics can quickly and neatly be explained in the algebraic context of predicate transformers. But then the algebra should have a standard model.

2 The Calculus of Flowsets

Let $\mathcal{S}$ be any set, the elements of which are called *states*. We will model computations by sequences of states, called 'execution sequences', or *exseqs* for short. Terminating computations are modelled by finite sequences, non-terminating ones by infinite sequences. Amongst terminating computations we distinguish those which terminate *cleanly* from those which don't. In the latter case we say the computation *aborts*, and we model this by having the exseq for that computation ending in a special symbol '$\perp$', with $\perp \notin \mathcal{S}$.

We define the set $\mathrm{Seq}(\mathcal{S})$ of exseqs as follows:

(1) $\mathcal{S}^+$ denotes the set of all finite non-empty sequences of elements of $\mathcal{S}$.

$\mathcal{S}^\perp$ denotes the set of all finite nonempty sequences with $\perp$ as the last component and elements of $\mathcal{S}$ as the first and (if any) intermediate components.

$\mathcal{S}^\infty$ denotes the set of all infinite sequences of elements of $\mathcal{S}$.

$\mathrm{Seq}(\mathcal{S}) = \mathcal{S}^+ \bigcup \mathcal{S}^\perp \bigcup \mathcal{S}^\infty$.

Note that $\mathcal{S}^+, \mathcal{S}^\perp$ and $\mathcal{S}^\infty$ are disjoint. These three sets naturally represent (respectively) cleanly terminating computations, computations terminating in abortion, and non-terminating computations. We denote exseqs by $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, ... etc, the idea being that when we write '$\mathbf{x} = (x_1, x_2, x_3, \ldots)$' it is left open whether or not $\mathbf{x}$ is finite. But when we write '$\mathbf{x} = (x_1, x_2, \ldots, x_n)$' then $\mathbf{x}$ *is* finite and has x_n as last element, where either $x_n \in \mathcal{S}$ (if $\mathbf{x} \in \mathcal{S}^+$) or $x_n = \perp$ (if $\mathbf{x} \in \mathcal{S}^\perp$).

We now define some operations on exseqs. For any $\mathbf{x} \in \mathrm{Seq}(\mathcal{S})$, say $\mathbf{x} = (x_1, x_2, x_3, \ldots)$:

(2) $first(\mathbf{x}) = x_1$

$$length(\mathbf{x}) = \begin{cases} n & \text{if } \mathbf{x} \in \mathcal{S}^+ \bigcup \mathcal{S}^\perp \text{ and } \mathbf{x} = (x_1, x_2, \ldots, x_n) \\ \infty & \text{if } \mathbf{x} \in \mathcal{S}^\infty \end{cases}$$

$$last(\mathbf{x}) = \begin{cases} x_n & \text{if } \mathbf{x} \in \mathcal{S}^+ \bigcup \mathcal{S}^\perp \text{ and } \mathbf{x} = (x_1, x_2, \ldots, x_n) \\ \text{undefined} & \text{if } \mathbf{x} \in \mathcal{S}^\infty \ . \end{cases}$$

We also need to be able to *compose* two exseqs $\mathbf{x}$ and $\mathbf{y}$. In the paradigm case this takes place when $\mathbf{x}$ terminates in exactly that state in which $\mathbf{y}$ begins: then $\mathbf{x} \circ \mathbf{y}$ is obtained by identifying $\mathbf{x}$'s last component with $\mathbf{y}$'s first component, thus joining the two exseqs together. For other cases special provision must be made. As follows:

(3) For any $\mathbf{x}, \mathbf{y} \in \text{Seq}(\mathcal{S})$,

$$\mathbf{x} \circ \mathbf{y} = \begin{cases} \mathbf{x} & \text{if } \mathbf{x} \in \mathcal{S}^\perp \bigcup \mathcal{S}^\infty \\ (x_1, x_2, \ldots, x_n, y_2, y_3, \ldots) & \text{if } \mathbf{x} \in \mathcal{S}^+, \text{ say } \mathbf{x} = (x_1, x_2, \ldots, x_n) \text{ and} \\ & \mathbf{y} = (y_1, y_2, y_3, \ldots), \text{ and } last(\mathbf{x}) = first(\mathbf{y}) \\ \text{undefined} & \text{otherwise} \ . \end{cases}$$

The idea is that if a computation aborts, or does not terminate, then formally sequencing another computation after it has no effect. Sequencing is only effective when the second computation picks up where the first left off.

Finally, we make explicit the usual *prefix ordering* of sequences — also called 'ordering by initial subsequences'. For any $\mathbf{x}, \mathbf{y} \in \text{Seq}(\mathcal{S})$, we have (with $n < \infty$ for every $n \in \mathcal{N}$):

(4) $\mathbf{x} \leq \mathbf{y}$ iff (i) $length(\mathbf{x}) \leq length(\mathbf{y})$, and

(ii) $x_i = y_i$ for every i such that $0 \leq i \leq length(\mathbf{x})$.

We can now explore some mathematical properties of our little calculus of exseqs. To begin with, $\circ$ is associative and $\leq$ is a partial order. Thus:

(5) **Lemma** *$(Seq(\mathcal{S}), \circ, \leq)$ is a partially ordered semigroup.*

As is often the case in dealing with partially ordered structures (e.g. in denotational semantics), it will be important that chains have least upper bounds. Recall that a *chain* C in a partially ordered set $(X, \leq)$ is a linearly ordered subset of X — i.e. one such that for any $x, y \in C$ either $x \leq y$ or $y \leq x$. We denote least upper bounds by the neutral notation 'lub'.

(6) **Lemma** *Every chain in Seq(S) has a least upper bound.*

Proof Let $\{\mathbf{x}_i\}_{i \in I}$ be a chain of exseqs (under the prefix ordering). Define $\mathbf{x}$ to be that exseq such that (i) $length(\mathbf{x}) = \text{lub}\{length(\mathbf{y}) \mid \mathbf{y} \in \{\mathbf{x}_i\}_{i \in I}\}$ and (ii) $\forall \mathbf{y} \in \{\mathbf{x}_i\}_{i \in I}$ and $\forall j : 0 \leq j \leq length(\mathbf{y})$ we have $y_j = x_j$. Intuitively, each $\mathbf{x}_i$ is a prefix of $\mathbf{x}_{i+1}$, and $\mathbf{x}$

is that exseq of which all the x_i's are prefixes. Also, $\text{lub}(\mathcal{N}) = \infty$. We leave it as an exercise to show that $\mathbf{x} = \text{lub}\ \{x_i\}_{i \in I}$. □

Exseqs will model computations, but our aim is more ambitious than that: we wish to model the programs from which these computations arise. We do so by using the *power construction* expounded (e.g.) in Goldblatt [8] and Grätzer and Whitney [17]. That is, from the partially ordered semigroup $(Seq(\mathcal{S}), \circ, \leq)$we can form its *power structure* $(\mathcal{P}(\text{Seq}(\mathcal{S})), \circ, \leftharpoonup, E)$, where:

(7) $\mathcal{P}(\text{Seq}(\mathcal{S}))$ is the set of all subsets $X, Y, Z, \ldots$ of $\text{Seq}(\mathcal{S})$;

$X \circ Y = \{\mathbf{x} \circ \mathbf{y} \mid \mathbf{x} \in X \text{ and } \mathbf{y} \in Y\} \quad \forall X, Y \in \mathcal{P}(\text{Seq}(\mathcal{S}))$;

$X \leftharpoonup Y$ iff $(\forall \mathbf{x} \in X)(\exists \mathbf{y} \in Y)[\mathbf{x} \leq \mathbf{y}]$ and $(\forall \mathbf{y} \in Y)(\exists \mathbf{x} \in X)[\mathbf{x} \leq \mathbf{y}]$,

$\forall X, Y \in \mathcal{P}(\text{Seq}(\mathcal{S}))$;

$E = \{(s) | s \in S\}$ (i.e. the set of one-component sequences).

[Note: Readers familiar with powerdomain theory in denotational semantics will recognise $\leftharpoonup$ as the *Egli-Milner ordering*.]

The required model of programs is obtained as a substructure of this power structure, namely that consisting of certain special sets of exseqs.

(8) A set $X \in \mathcal{P}(\text{Seq}(\mathcal{S}))$ of exseqs is called a *flowset* if it satisfies

Axiom 1: $\forall s \in \mathcal{S}\ \exists \mathbf{x} \in X$ with $first(\mathbf{x}) = s$, and

Axiom 2: $\forall\ \mathbf{x}, \mathbf{y} \in X, \mathbf{x} \not\leq \mathbf{y}$.

The set of all flowsets will be denoted by '$\mathcal{F}$'.

The idea with Axiom 1 is that any state s is a possible initial state of any program. A computation may not actually progress from s (in which case it is modelled by the exseq (s)), or it may immediately abort (modelled by the exseq $(s, \perp)$) — but at any rate it is *defined*. One virtue of this idea is that, for any state s, any set of exseqs $\mathbf{x}$ with $first(\mathbf{x}) = s$ provides a natural model for the *execution tree* (or 'extree' for short) of some program α from this intial state s. Note that it is risky to rely on a graphical presentation of such trees, or a definition intended to capture such a graphical presentation. For example, a program α may, from some given initial state s, have the possible computation sequences (s, t_1, u, v_1) and (s, t_2, u, v_2), and only these. Yet the graphical tree representation of Figure 1 would seem to indicate that (s, t_1, u, v_2) and (s, t_2, u, v_1) are also possible execution sequences. But this is not intended.

The idea with Axiom 2 is that no initial subsequence of an execution sequence is also an

execution sequence. This, in fact, is the manifestation of a rather subtle point concerning

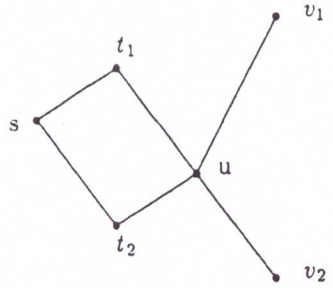

Figure 1

nondeterminism. Generally speaking, a program is said to be nondeterministic if from any given state there is no assumption of a *unique* next state. The question raised here, which does not seem to have been addressed before, is whether or not one should uniformly assume the *existence* of a next state. In short: does the notion of nondeterminism allow a program which has already proceeded up to some state s to sometimes terminate at s and sometimes not? Space does not allow a discussion of this issue here, so we resolve the matter by simply choosing one of the alternatives: Axiom 2 rules out a notion of nondeterminism on which initial subsequences of computations can also be computations. Note that any extree satisfies Axiom 2 (but not necessarily Axiom 1).

The operation $\circ$ between flowsets is intended to model sequential composition. Think of X and Y respectively as the set of all possible computations of programs α and β. Then $X \circ Y$ corresponds to the set of all possible computations of the program which consists of doing β immediately after α. Technically: for any sequence $\mathbf{x} \in X$, if it does not terminate cleanly leave it; if it does then append at its final state (say) x_n all exseqs in Y starting with x_n. The relation $\eqslantless$ between flowsets is the power order of the prefix relation between exseqs. It says that $X \eqslantless Y$ iff any exseq in X is a prefix of some exseq in Y, and any exseq in Y has some exseq in X as prefix. Thus, intuitively, Y *extends* X. The set E is useful for technical reasons: it will model a program called *null*, which from any initial state s does exactly nothing.

We now come to the calculus of flowsets.

(9) **Theorem** $(\mathcal{F}, \circ, E, \eqslantless)$ *is a partially ordered monoid, with* E *as identity for* $\circ$, *and as minimum under* $\eqslantless$.

Proof To establish the monoid part of the Theorem we need to check that the power operation $\circ$ is associative, and that E is a (left- and right-) identity for $\circ$. The former follows since the power operation of any associative operation is again associative. The

latter is also easy, but it is not immediate. That $X \circ E = X$ and $E \circ X = X$ for any flowset X must be checked for all possible cases.

Secondly we must establish that $\eqsim$ is a partial order, and E its minimum. It is known that the power order of any partial order is a quasi-order — that is, reflexive and transitive. So we only need to check that $\eqsim$ is anti-symmetric. To do so, let X and Y be flowsets such that $X \eqsim Y$ and $Y \eqsim X$; we need to show that $X = Y$. For this we show that $X \subseteq Y$, analogous reasoning would then also establish that $Y \subseteq X$. So let $\mathbf{x} \in X$ arbitrarily. Since $X \eqsim Y$ there is some $\mathbf{y} \in Y$ such that $\mathbf{x} \leq \mathbf{y}$. But also, since $Y \eqsim X$, there is then some $\mathbf{z} \in X$ such that $\mathbf{y} \leq \mathbf{z}$. But then $\mathbf{x} \leq \mathbf{z}$, hence by Axiom 2 $\mathbf{x} = \mathbf{z}$, hence since $\mathbf{x} \leq \mathbf{y} \leq \mathbf{z}$ we have $\mathbf{x} = \mathbf{y}$, hence $\mathbf{x} \in Y$. Thus $X \subseteq Y$, as required. To see that E is the minimum element of $\eqsim$ we just need to invoke Axiom 1. □

The next question to address is whether chains in $\mathcal{F}$ have least upper bounds. Note that the 'obvious' response that flowsets are sets and that therefore lub's should be unions is fallacious. The union of flowsets is indeed a set, but it is not necessarily a flowset: Axiom 2 is easily violated. However, our earlier preparation in Lemma (6) pays off: chains of flowsets will have lub's because chains of exseqs have them.

(10) **Theorem** *Let $\{A_i\}_{i \in I}$ be any chain of flowsets under the ordering $\eqsim$. Let*

$$A = \{\mathbf{x} \mid \mathbf{x} = \textit{lub under } \leq \textit{ of some chain } C \textit{ in } \bigcup_{i \in I} A_i\}.$$

Then $A = \textit{lub } \{A_i\}_{i \in I} \textit{ under } \eqsim$.

In §3 we will view the iterative command DO as a repetition of IF commands. We therefore need to build into the calculus of flowsets the notion of repetition; it is precisely for this purpose that we talked above about chains and lub's. We define *iterated composition* in the usual way:

(11) For any flowset X: $X^0 = E$

$$X^{n+1} = X^n \circ X, \quad \forall n \geq 0.$$

Then:

(12) **Theorem** *For any flowset X, $\{X^n\}_{n \geq 0}$ forms a chain under the ordering $\eqsim$.*

Proof We must show that $X^0 \eqsim X^1 \eqsim X^2 \eqsim \ldots$. By Theorem (9) $E \eqsim X$ for any flowset X, so we only need to show that, $\forall n \geq 0$, $X^n \eqsim X^{n+1} = X^n \circ X$. Since composition simply appends exseqs in X to exseqs in X^n this is easy to show by consideration of cases. □

It now follows from Theorem (10) that for any flowset X the chain $\{X^n\}_{n \geq 0}$ must have a

lub. This is an important notion, for which we reserve both a notation and a name.

(13) For any flowset X, the *iteration* X^* of X is defined by $X^* = \text{lub}\{X^n\}_{n\geq 0}$.

Finally we introduce into the calculus of flowsets an operator which does not explicitly feature in the Gries/Dijkstra algebra of weakest preconditions, but which is quite useful as an aid in modelling such operations. It is the *nondeterministic choice operator* which for programs α and β would correspond to a program $\alpha \vee \beta$, interpreted as: 'Nondeterministically choose either α or β and then run the chosen program'. In the relational semantics briefly discussed in §2 nondeterministic choice is modelled by set-theoretic union. But in the calculus of flowsets this won't do, since the union of two flowsets need not be a flowset. Instead we choose to model $\alpha \vee \beta$ by what is known as *Hilbert's epsilon operator.*

This operator, ϵ, features strongly in Higher-Order Logic — see for example [1], [9], and [10]. It is a variable-binding operator, like quantifiers, and can be used as a primitive logical symbol. For our purposes, however, it will suffice to make clear its semantics. Namely, the ϵ-operator acts as a *choice function*: given any set A, ϵ picks out some unknown but fixed element of A, which is then denoted by $\epsilon.A$. In particular,

(14) For any indexed set $\{X_i\}_{i\in I}$ of flowsets

$$\epsilon.\{X_i\}_{i\in I}$$

denotes some particular unspecified but fixed X_i, $i \in I$.

Note that if the indexed set I is finite then $\epsilon.\{X_1, X_2, \ldots, X_n\}$ is some particular one of n flowsets. But we make no finiteness constraints on I in (14), in order to cater for the unbounded nondeterminism of Dijkstra and Scholten [6].

In conclusion we point out some related work. Blikle [3] also models programs as sets of computations and presents an algebra of such sets. But Blikle adopts a notion of 'generalised composition', whereas our approach uses the power construction. Another relevant paper is Hoare [14], which models programs as sets of 'possible traces', along the lines of operational semantics. It is interesting that Hoare ([14] p425) proves exactly what we called Axioms 1 and 2 for flowsets.

3 Verifying the Gries/Dijkstra Conditions

Our aim in this section should be clear, and is easy to state. We model each of the programs in the Gries/Dijkstra language by a flowset, each operation on programs by an operation

on flowsets, and we prove the given formulae of the algebra of weakest preconditions in the calculus of flowsets. We also add a few extra features to the Gries/Dijkstra algebra, as an aid to the exposition. (No substantive changes are made).

We adopt the square bracket notation '$[\![\cdot]\!]$' of denotational semantics to map each program α onto its *meaning* $[\![\alpha]\!]$, which will be a flowset. We may also specialise this to 'the meaning of a program α at some initial state s', written $[\![\alpha]\!](s)$, which is an *extree* (or execution tree) in the sense of §2: the set of all those exseqs $\mathbf{x}$ in $[\![\alpha]\!]$ such that first($\mathbf{x}$)= s.

To begin with:

(1) $[\![skip]\!] = \{(s,s)|s \in \mathcal{S}\}$

$[\![abort]\!] = \{(s,\perp)|s \in \mathcal{S}\}$

$[\![havoc]\!] = \{(s,t)|s \in \mathcal{S} \text{ and } t \in \mathcal{S}\}$

$[\![null]\!] = \{(s)|s \in \mathcal{S}\} = E$

We introduce the atomic program *null* because it will be useful in defining IF. Next, we model sequential composition of programs by composition of flowsets. That is:

(2) For any programs α and β, $[\![\alpha;\beta]\!] = [\![\alpha]\!] \circ [\![\beta]\!]$.

The assignment statement, as is well known, is problematic in the Gries/Dijkstra algebra insofar as it is the only command which deals directly with program variables. But the problems raised by assignment are extraneous to the modelling proposed here, and does not affect the issues we discuss. We therefore simply adopt the Gries/Dijkstra technique (also commonplace in other contexts) of indicating notationally a change in state effected by a change in the value of a program variable. Namely:

(3) For any state $s \in \mathcal{S}$, '$s[e/z]$' will denote that state which differs from s only in that the value of the program variable z is replaced by the value of the expression e evaluated in s.

It is then easy to model the assignment statement as a flowset. [Note: Like Gries ([12] p118), we assume e to be well-defined in every S.]

(4) For any program variable z and expression e,

$$[\![z := e]\!] = \{(s, s[e/z])|s \in \mathcal{S}\}$$

To model IF we use both the Hilbert epsilon operator and the null command. We first define nondeterministic choice:

(5) For any programs α and β, $[\![\alpha \vee \beta]\!] = \epsilon.\{[\![\alpha]\!], [\![\beta]\!]\}$.

We then define the meaning of IF at an arbitrary initial state s, thus obtaining an extree, and take the union of all of these extrees. As follows:

(6) For any programs α_1 and α_2, any predicates B_1 and B_2, and any state $s \in S$, the *meaning of* '$if\ B_1 \to \alpha_1\ []\ B_2 \to \alpha_2\ fi$' *at* s is given by:

$$[\![if\ B_1 \to \alpha_1\ []\ B_2 \to \alpha_2\ fi]\!](s) = \begin{cases} [\![\alpha]\!](s) & \text{if } s \in B_1 \text{ and } s \notin B_2 \\ [\![\beta]\!](s) & \text{if } s \notin B_1 \text{ and } s \in B_2 \\ [\![\alpha \vee \beta]\!](s) & \text{if } s \in B_1 \text{ and } s \in B_2 \\ [\![null]\!](s) & \text{if } s \notin B_1 \text{ and } s \notin B_2 \end{cases}$$

and the meaning of IF itself is:

$$[\![if\ B_1 \to \alpha_1\ []\ B_2 \to \alpha_2\ fi]\!] = \bigcup_{s \in S} [\![if\ B_1 \to \alpha_1\ []\ B_2 \to \alpha_2\ fi]\!](s).$$

[Note: We use, as is customary in this context, '$s \in B$' as an abbreviation for 'B is true in state s'. Likewise '$s \notin B$' means 'B is false in s'.] The intention here should be clear. How does '$if\ B_1 \to \alpha_1\ []\ B_2 \to \alpha_2\ fi$' execute from any initial state s? If B_1 is true but B_2 is not α_1 will be executed from s; if B_1 is false but B_2 is true then α_2 will be executed; if both are true exactly one of α_1 and α_2 will be nondeterministically selected and executed, and if neither B_1 nor B_2 is true nothing will happen. Two typical special cases of IF are '$if\ B\ do\ \alpha$', which we equate to '$if\ B \to \alpha\ []\ \neg B \to null\ fi$', and $if\ B\ do\ \alpha\ else\ \beta\ fi$', which we equate to '$if\ B \to \alpha\ []\ \neg B \to \beta$'. It is then easy to read off their meanings from (6) above.

With Gries and Dijkstra we assume the guards to be well-defined in every state, so that the four possibilities enumerated above are the only ones. On the other hand, unlike Gries ([12] p132), Dijkstra ([5] p34) and Dijkstra and Scholten ([6] p144) we do *not* say that if no guard is true then IF aborts. There is danger of confusion here, since in fact in Dijkstra ([5] p26) and Dijkstra and Scholten ([6] p135) 'abort' really means 'does not terminate', so what they are actually saying is that if no guard is true then IF goes into an endless and unproductive loop. It is not clear to us what the virtues are of such a notion. What will be made clear by our exposition, we trust, is that having (literally) the *null*-option available makes for a tidy treatment of IF, both technically and conceptually. On our treatment, in any composition '$IF;\alpha$' control will always pass from IF to α, even when no guard of IF is true. The reader is reminded that our aim is to model the *algebra* of weakest preconditions, and that in doing so we are not constrained by any particular intuitive *semantics* of the constructs involved. A final point: each extree by assumption satisfies Axiom 2 (of §2), hence a union of extrees

over *every* state $s \in \mathcal{S}$ will be a flowset. Thus $[\![if\ B_1 \to \alpha_1\ []\ B_2 \to \alpha_2\ fi]\!]$ is well-defined as a flowset.

For the iterative command DO, in the simple form '*while B do* α', our earlier preparation really pays off. The definition is quite simple: DO is the iteration of IF.

(7) $[\![while\ B\ do\ \alpha]\!] = [\![if\ B\ do\ \alpha]\!]^*$ $(= \text{lub}\ \{[\![if\ B\ do\ \alpha]\!]^n\}_{n\geq 0} = \text{lub}\{[\![(if\ B\ do\ \alpha)^n]\!]\}_{n\geq 0})$

Recalling the relevant definitions in §2, we see that DO is defined as the least upper bound of the chain of flowsets arising from repeating the IF command. [Note: This is the second equality in (7). The third is easy to prove by induction.] Intuitively, to perform '*while B do* α' consists of repeatedly doing the following, until it has no further effect: check whether B is true, and if so do α.

We now come to the central definition, which is that of weakest precondition: for any program α and predicate Q, Dijkstra ([5] p16,17), Gries ([12] 7.1 p108) and Dijkstra and Scholten ([6] p129) are unanimous that '$wp(\alpha, Q)$' must denote the set of all those states such that execution of α begun in one of them is guaranteed to terminate, and when it does it satisfies Q. Our definition will capture this intuition, but it adds a clarification: '$wp(\alpha, Q)$' will denote the set of all states from which α terminates *cleanly* (and satisfies Q upon termination). In our context, if α does not terminate cleanly (i.e. aborts) it cannot satisfy Q, since $\perp$ is not a state.

(8) For any program α and predicate Q,

$$wp(\alpha, Q) = \{s \in \mathcal{S} | (\forall \mathbf{x} \in [\![\alpha]\!](s))[\mathbf{x} \in \mathcal{S}^+ \text{ and } last(\mathbf{x}) \in Q]\}$$

We can now verify the formulae of §1.

(9) **Theorem** *For any program* α *and predicates* Q *and* R*:*

(a) (Gries [12], 7.3) **Law of the Excluded Miracle:** $wp(\alpha, \emptyset) = \emptyset$.

(b) (Gries [12], 7.4) **Distributivity of Conjunction:**

$$wp(\alpha, Q) \bigcap wp(\alpha, R) = wp(\alpha, Q \bigcap R).$$

(c) (Gries [12], 7.5) **Law of Monotonicity:**

$$\text{If } Q \subseteq R \text{ then } wp(\alpha, Q) \subseteq wp(\alpha, R).$$

(d) (Gries [12], 7.6) **Distributivity of Disjunction:**

$$wp(\alpha, Q) \bigcup wp(\alpha, R) \subseteq wp(\alpha, Q \bigcup R).$$

Proof All of these depend upon simple logical properties, such as in (b) the distribution of universal quantification over conjunction. □

To check that in our context the converse of Theorem (9)(d) does not hold in general an example such as that of Gries ([12] p111) would suffice. But the converse *does* hold for deterministic programs.

(10) A program α is said to be *deterministic* iff for every $s \in \mathcal{S}$ $[\![\alpha]\!](s)$ is a singleton set.

That is, from any initial state α can proceed to execute in exactly one way.

(11) **Theorem** *For any predicates Q and R, and any deterministic program α*
(Gries [12] 7.7) $wp(\alpha, Q) \bigcup wp(\alpha, R) = wp(\alpha, Q \bigcup R)$.

The atomic programs are easy to characterise from Definition (1).

(12) **Theorem** *For any predicate Q*

(a) (Gries [12], 8.1)	$wp(skip, Q)$	$= Q$.
(b) (Gries [12], 8.2)	$wp(abort, Q)$	$= \emptyset$.
(c) (Dijkstra and Scholten [6], 7.12)	$wp(havoc, Q)$	$= \mathcal{S}$
(d)	$wp(null, Q)$	$= Q$.

We now come to composition: the place where the relational model fails.

(13) **Theorem** *For any programs α and β, and any predicate Q,*

$$wp(\alpha;\beta, Q) = wp(\alpha, wp(\beta, Q)).$$

Proof Left to right: Let $s \in wp(\alpha;\beta, Q)$, then by (8) any $\mathbf{z} \in [\![\alpha;\beta]\!](s)$ terminates cleanly, and in Q. To show that $s \in wp(\alpha, wp(\beta, Q))$, let $\mathbf{u} \in [\![\alpha]\!](s)$ arbitrarily. If $\mathbf{u} \in \mathcal{S}^{\perp}$ or $\mathbf{u} \in \mathcal{S}^{\infty}$ then also by (2.3) (and (2)) we would have $\mathbf{u} \in [\![\alpha;\beta]\!](s)$, which would then contradict the fact that $\mathbf{u}$ must terminate cleanly. So $\mathbf{u} \in \mathcal{S}^{+}$, hence $last(\mathbf{u}) \in \mathcal{S}$. To show that $last(\mathbf{u}) \in wp(\beta, Q)$, consider any $\mathbf{v} \in [\![\beta]\!](last(\mathbf{u}))$. Then $\mathbf{u} \circ \mathbf{v} \in [\![\alpha;\beta]\!](s)$, hence by assumption $\mathbf{u} \circ \mathbf{v}$ terminates cleanly, and in Q. But then $\mathbf{v}$ must terminate cleanly, and in Q. Hence $last(\mathbf{u}) \in wp(\beta, Q)$, as required. Analogous reasoning establishes the right to left inclusion. □

As with the definition of the assignment statement we also pass lightly over its weakest precondition result: issues such as definability and non-classical conjunction raised by Gries ([12] 9.1.1) are not germane to our discussion. What we should do is check:

(14) **Theorem** *For any program variable z, expression e and predicate Q:*

$$wp(\text{`}z := e\text{'}, Q) = \{s \mid s[e/z] \in Q\}.$$

Proof Let $s \in wp(\text{`}z := e\text{'}, Q)$ then every $\mathbf{x} \in [\![z := e]\!](s)$ terminates cleanly and in Q. But by definition (4) $\mathbf{x} = (s, s[e/z])$; hence $s[e/z] \in Q$. For the reverse direction, let s be

such that $s[e/z] \in Q$. To show that $s \in wp(\text{`}z := e\text{'},Q)$ take any $\mathbf{x} \in [\![z := e]\!](s)$, then by definition (4) $\mathbf{x} = (s, s[e/z])$. Hence $\mathbf{x}$ terminates cleanly and in $Q$; so $s \in wp(\text{`}z := e\text{'},Q)$. □

Since we use the (nondeterministic) choice operator in our modelling of IF we determine also the weakest precondition for $\vee$. It is quite neat:

(15) **Theorem** *For any programs α and β, and any predicate Q,*

$$wp(\alpha \vee \beta, Q) = wp(\alpha, Q) \bigcap wp(\beta, Q).$$

Proof Left to right: Let $s \in wp(\alpha \vee \beta, Q)$. Then $\forall \mathbf{x} \in [\![\alpha \vee \beta]\!](s)$ we must have that $\mathbf{x}$ terminates cleanly and in Q. But $[\![\alpha \vee \beta]\!](s)$ can be either $[\![\alpha]\!](s)$ or $[\![\beta]\!](s)$, which means that every $\mathbf{x}$ in $[\![\alpha]\!](s)$ must terminate cleanly and in Q, *and* so must every $\mathbf{x}$ in $[\![\beta]\!](s)$. Hence $\mathbf{x} \in wp(\alpha, Q) \bigcap wp(\beta, Q)$. Right to left: Similar. □

For the IF operator we are required to verify Gries ([12] 10.3). Aiming in this direction, we get:

(16) **Theorem** *For any programs α and β, and any predicates B_1, B_2 and Q,*

$wp(if\ B_1 \rightarrow \alpha_1 \ [\!]\ B_2 \rightarrow \alpha_2\ fi, Q)$

$= [(B_1 \bigcup B_2) \bigcap (\neg B_1 \bigcup wp(\alpha_1, Q)) \bigcap (\neg B_2 \bigcup wp(\alpha_2, Q))] \bigcup [\neg B_1 \bigcap \neg B_2 \bigcap Q]$.

[Note: '$\neg B$' denotes 'not B', or (set-theoretically) the complement of B].

Proof Left to right: Let $s \in wp(if\ B_1 \rightarrow \alpha_1 \ [\!]\ B_2 \rightarrow \alpha_2\ fi, Q)$. Then every $\mathbf{x} \in [\![if\ B_1 \rightarrow \alpha_1 \ [\!]\ B_2 \rightarrow \alpha_2\ fi]\!](s)$ terminates cleanly and in Q. To show s is in the right hand side we consider two cases: either $s \in B_1 \bigcup B_2$ or $s \in \neg(B_1 \bigcup B_2)$. The latter is easy for then by definition (6) $[\![if\ B_1 \rightarrow \alpha_1 \ [\!]\ B_2 \rightarrow \alpha_2\ fi]\!](s) = [\![null]\!](s)$; hence $s \in \neg B_1 \bigcap \neg B_2 \bigcap Q$. In the former case either $s \in B_1$ or $s \notin B_1$. If $s \in B_1$ then $[\![if\ B_1 \rightarrow \alpha_1 \ [\!]\ B_2 \rightarrow \alpha_2\ fi]\!](s) = [\![\alpha_1]\!](s)$; hence $s \in wp(\alpha_1, Q) \subseteq \neg B_1 \bigcup wp(\alpha_1, Q)$. On the other hand if $s \in \neg B_1$ then evidently $s \in \neg B_1 \bigcup wp(\alpha_1, Q)$ thus $s \in \neg B_1 \bigcup wp(\alpha_1, Q)$ in any case. Similarly, $s \in \neg B_2 \bigcup wp(\alpha_2, Q)$. Hence $s \in [B_1 \bigcup B_2] \bigcap [\neg B_1 \bigcup wp(\alpha_1, Q)] \bigcap [\neg B_2 \bigcup wp(\alpha_2, Q)]$. Right to left: By a similar consideration of all possible cases. □

To read this theorem colloquially, consider this: either some guard in IF is true, or no guard is. In the former case $wp(\text{IF},Q)$ is an intersection of three facts: some guard is true, if B_1 is true then we have $wp(\alpha_1, Q)$, and if B_2 is true then we have $wp(\alpha_2, Q)$. This is an exact transcript of Gries ([12] 10.3). But as pointed out earlier, our treatment differs from that of Gries/Dijkstra in also covering explicitly the case where no guard is true: in that case

wp(IF,Q) is just Q. It seems to us that this better captures the intuition behind IF than the Gries/Dijkstra idea that IF is non-terminating when no guard is true.

But there is yet a better way of expressing wp(IF,Q):

(17) **Theorem** *For any programs α and β, and any predicates B_1, B_2, and Q,*

$wp(if\ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2\ fi, Q)$

$= [B_1 \bigcap \neg B_2 \bigcap wp(\alpha_1, Q)] \bigcup [\neg B_1 \bigcap B_2 \bigcap wp(\alpha_2, Q)] \bigcup [B_1 \bigcap B_2 \bigcap wp(\alpha_1 \vee \alpha_2, Q)]$

$\bigcup [\neg B_1 \bigcap \neg B_2 \bigcap Q]$.

Proof Left to right: Let $s \in wp(if\ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2\ fi, Q)$. Then every $\mathbf{x} \in [\![if\ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2\ fi]\!](s)$ terminates cleanly and in Q. To show s is in the right hand side we distinguish four mutually exclusive and jointly exhaustive cases: either $s \in B_1 \bigcap \neg B_2$ or $s \in \neg B_1 \bigcap B_2$ or $s \in B_1 \bigcap B_2$ or $s \in \neg B_1 \bigcap \neg B_2$. We only consider the first case; the others are similar: If $s \in B_1 \bigcap \neg B_2$ then by definition (6) $[\![if\ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2\ fi]\!](s) = [\![\alpha_1]\!](s)$; hence every $\mathbf{x} \in [\![\alpha]\!](s)$ terminates cleanly and in Q. So $s \in B_1 \bigcap \neg B_2 \bigcap wp(\alpha_1, Q)$. Right to left: Similar. □

This presentation of wp(IF,Q) exactly parallels the definition of IF in (6). That is, wp(IF,Q) breaks down as follows: If the first guard is true and the second false we are dealing with $wp(\alpha_1, Q)$; if the first guard is false and the second true we are dealing with $wp(\alpha_2, Q)$; if both guards are true we are dealing with the weakest precondition of *one* of α_1 or α_2 (without knowing which), and if no guard is true the weakest precondition is Q itself. In thus explicitly presenting all possible cases the formulation is superior to that of (16). [Note: It is a nontrivial exercise to show directly that the two formulations are equivalent.]

For '*if B do α*' and '*if B do α else β fi*' we get as special cases from (17):

(18) **Corollary** *For any programs α and β, and any predicates B and Q,*

$wp(if\ B\ do\ \alpha, Q) = [B \bigcap wp(\alpha, Q)] \bigcup [\neg B \bigcap Q]$

$wp(if\ B\ do\ \alpha\ else\ \beta\ fi, Q) = [B \bigcap wp(\alpha, Q)] \bigcup [\neg B \bigcap wp(\beta, Q)]$ □

As Gries ([12] p135) points out we are often not interested in the weakest precondition of IF, but only in determining if a known precondition implies it. Gries's Theorem (10.5) gives necessary and sufficient conditions under which $X \subseteq wp$(IF,Q) is true. The diligent reader will have no problem with proving this theorem in our context.

Finally, we come to the iterative command, '*while B do α*'. Like Gries ([12] p140) we define for any given predicate Q a sequence $H_n(Q)$ of predicates, where $H_n(Q)$ represents the set of all states from which execution of DO terminates (cleanly!) in n or fewer iterations,

with Q true. But our definition simplifies that of Gries. Namely:

(19) For '*while* B *do* α' define predicates $H_n(Q)$, $n \geq 0$, by:

$H_0(Q) = \neg B \cap Q$

$H_{n+1}(Q) = wp(if\ B\ do\ \alpha, H_n(Q)), \quad \forall n \geq 0.$

The simplification is possible because of our treatment of IF in the case where no guards are true. To prove that nothing is omitted by the simplification we require two Lemmas.

(20) **Lemma** $\neg B \cap H_n(Q) = \neg B \cap Q, \quad \forall n \geq 0.$

Proof By induction. □

This says that any state from which DO terminates in n or fewer iterations, and in which B is false, is also a state in which Q is true.

(21) **Lemma** $H_{n+1}(Q) = [\neg B \cap Q] \cup [B \cap wp(\alpha, H_n(Q))], \quad \forall n \geq 0.$

Proof Use (19),(18) and (20). □

This gives exactly the *form* of Gries's $H_{n+1}(Q)$. An inductive argument then suffices to show that it is also the same *set*. So we get:

(22) **Theorem** *$H_n(Q)$ as defined by Gries ([12] p140) on the basis of his definition (10.3b, p132) of wp(IF,Q) is the same set $\forall n$ as $H_n(Q)$ defined in definition (19) on the basis of wp(IF,Q) given in Theorem (16) (or (17)), arising from our definition (6) of IF.*

It remains to verify Gries ([12] 11.2).

(23) **Lemma** *For 'while B do α', and any predicate Q, $\{H_n(Q)\}_{n \geq 0}$ forms a chain under the set-theoretic ordering $\subseteq$.*

This means that Gries really characterises $wp(DO, \alpha)$ as the least upper bound of the chain (under $\subseteq$) of the $H_n(Q)$'s. And so do we. Again we need two Lemmas. Both demonstrate our approach of defining DO in terms of IF.

(24) **Lemma** $H_n(Q) = wp((if\ B\ do\ \alpha)^n, \neg B \cap Q), \quad \forall n \geq 0.$

Proof By induction. [Note: For any program α, $\alpha^0 = null$ and $\alpha^{n+1} = \alpha^n; \alpha$.] □

(25) **Lemma** *For any $s \in S$, if $[\![while\ B\ do\ \alpha]\!](s)$ is finite then $\exists n \in \mathcal{N}$ such that $[\![while\ B\ do\ \alpha]\!](s) = [\![(if\ B\ do\ \alpha)^n]\!](s)$, and $t \notin B$ for every leaf t of this extree.*

Proof By saying 'the tree is finite' we mean 'only has branches of finite length'. (Since we are dealing with unbounded nondeterminism there may well be infinitely many branches).

From (7) we get that $[\![while\ B\ do\ \alpha]\!] = \text{lub}\{[\![(if\ B\ do\ \alpha)^n]\!]\}_{n\geq 0}$, hence if for any particular $s \in \mathcal{S}$ $[\![while\ B\ do\ \alpha]\!](s)$ is finite it follows from Theorem (2.10) that there must be a least number $m \in \mathcal{N}$ such that

$$[\![(if\ B\ do\ \alpha)^m]\!](s) = [\![(if\ B\ do\ \alpha)^{m+1}]\!](s) = \ldots$$

But then $[\![while\ B\ do\ \alpha]\!](s) = [\![(if\ B\ do\ \alpha)^m]\!](s)$, and $t \notin B$ (since otherwise $[\![(if\ B\ do\ \alpha)^{m+1}]\!](s)$ would extend $[\![(if\ B\ do\ \alpha)^m]\!](s)$). □

The idea here is quite simple: a terminating DO from an initial state s is precisely the n-fold composition of an IF-statement, for some $n \in \mathcal{N}$.

(26) **Theorem** (Gries [12] 11.2) *For any program α and predicates B and Q,*
$wp(while\ B\ do\ \alpha, Q) = \bigcup_{n\geq 0} H_n(Q) = \bigcup_{n\geq 0} wp((if\ B\ do\ \alpha)^n, \neg B \cap Q)$.
Proof Left to right: Let $s \in wp(while\ B\ do\ \alpha, Q)$. Then every exseq **x** in $[\![while\ B\ do\ \alpha]\!](s)$ terminates cleanly and in Q. Hence $[\![while\ B\ do\ \alpha]\!](s)$ is finite, so by Lemma (25) $\exists m \in \mathcal{N}$ such that $[\![while\ B\ do\ \alpha]\!](s) = [\![(if\ B\ do\ \alpha)^m]\!](s)$ (and $t \notin B$ for every leaf t of this extree). But then every exseq $\mathbf{x} \in [\![(if\ B\ do\ \alpha)^m]\!](s)$ terminates cleanly and in Q, hence in $\neg B \cap Q$. Thus by Lemmas (24) and (23), $s \in wp((if\ B\ do\ \alpha)^m, \neg B \cap Q) = H_m(Q) \subseteq \bigcup_{n\geq 0} H_n(Q)$.
Right to left: Let $s \in \bigcup_{n\geq 0} wp((if\ B\ do\ \alpha)^n, \neg B \cap Q)$, say $s \in wp((if\ B\ do\ \alpha)^m, \neg B \cap Q)$ for some $m \in \mathcal{N}$. Then every exseq $\mathbf{x} \in [\![(if\ B\ do\ \alpha)^m]\!](s)$ terminates cleanly and in $\neg B \cap Q$. But then

$$[\![(if\ B\ do\ \alpha)^m]\!](s) = [\![(if\ B\ do\ \alpha)^{m+1}]\!](s) = \ldots$$

and hence $[\![(if\ B\ do\ \alpha)^m]\!](s) = (\text{lub}\{[\![(if\ B\ do\ \alpha)^n]\!]\}_{n\geq 0})(s) =$ $[\![while\ B\ do\ \alpha]\!](s)$ by (7). Hence every **x** in $[\![while\ B\ do\ \alpha]\!](s)$ terminates cleanly and in Q and so $s \in wp(while\ B\ do\ \alpha, Q)$. □

4 Invariants

With DO, as with IF, the weakest precondition is not always the most useful precondition. As Gries ([12] p140) puts it:

> The formal definition of DO is not easy to use, and gives no insight into developing programs. Therefore, we want to develop a theorem that allows us to work with a useful precondition of a loop (with respect to a postcondition) that is not the weakest precondition.

This sought-after precondition is to be called an *invariant* of the loop: it is '... a predicate P that is true before and after each iteration of [the] loop' (Gries [12] p141). So the idea is that if $s \in P$ and DO is executed from s, then the final state is again an element of P. To model the notion of an invariant in our context we come forward with two suggestions.

(1) **Suggestion 1** Instead of restricting the notion of an invariant to loops, define it for any program α.

That is, for any program α a predicate $I \subseteq \mathcal{S}$ will be called an invariant of α iff, $\forall s \in I$, if α is executed from s then every final state is again an element of I. The reader will no doubt see the immediate problem: α may not terminate cleanly, or may not terminate at all, so that an appropriate final state may not exist. For this we have:

(2) **Suggestion 2** Think of invariants by analogy with subalgebras: 'I is an invariant of α' is analogous to a subset of an algebra being *closed* under a given operation.

The virtue of this suggestion is that the problem just raised has been exhaustively investigated in Universal Algebra, and so we may borrow from there. Since α may not terminate (cleanly, or at all), we may think of it as analogous to a *partial operation* in an algebra. The question of how to define invariants for programs which do not terminate cleanly is then analogous to this: *What is the correct notion of subalgebra for partial algebras?* For this, consider the comment of Grätzer ([11] p79):

> For algebras there is ony one reasonable way to define the concepts of subalgebra, homomorphism and congruence relation. For partial algebras we will define three different types of subalgebra, three types of homomorphism, and two types of congruence relation ... all of these concepts have their merits and their drawbacks, and each particular situation determines which one should be used.

Space constraints disallow further discussion, so we simply report that of the three kinds of subalgebra considered by Grätzer we may use two in the present context to give the following alternative notions of an invariant of a program α.

(3) **Alternative 1** A predicate I is called an *invariant* of a program α iff $\forall s \in I$ the extree $[\![\alpha]\!](s)$ is finite and all leaves are $\in I$.

(4) **Alternative 2** A predicate I is called an *invariant* of a program α iff $\forall s \in I$, *if* the extree $[\![\alpha]\!](s)$ is finite *then* all its leaves are $\in I$.

In the Gries/Dijkstra formulation of invariants termination is not built in — it must be proved separately by a bound function. Thus Gries/Dijkstra implicitly select Alternative 2, hence, for current purposes, so do we.

Having defined the notion of invariant, why is it useful? The idea is that an invariant, as a precondition of a loop, is easier to obtain than the weakest precondition. Namely, for '*while* B *do* α' and a given postcondition Q, if we can find an invariant I such that $\neg B \bigcap I \subseteq Q$, then I will be a precondition of the loop — that is, $I \subseteq wp(while\ B\ do\ \alpha, Q)$. The reasoning is that if execution is started in I it remains in I; upon termination B is false (hence $\neg B$ is true). But then any final state is in $\neg B \bigcap I$, hence in Q. This crops up in Dijkstra ([5] p38) as 'The Basic Theorem for the Repetitive Construct' (also 'The Fundamental Invariance Theorem for Loops'), in Gries ([12] p144) as Theorem (11.6) ('a theorem concerning a loop, an invariant and a bound function') and in Dijkstra and Scholten ([6] p180) as the 'Main Repetition Theorem'. Our version is:

(5) **Theorem** *For any predicates I, B and Q, and any program α, if*

(a) $\neg B \bigcap I \subseteq Q$, and

(b) $B \bigcap I \subseteq wp(\alpha, I)$, and

(c) $[\![while\ B\ do\ \alpha]\!](s)$ *is finite,* $\quad \forall s \in I$,

then

$$I \subseteq wp(while\ B\ do\ \alpha, Q).$$

Proof It suffices to show that $I \subseteq wp(while\ B\ do\ \alpha, I)$ since by (a) and monotonicity of wp, $wp(while\ B\ do\ \alpha, \neg B \bigcap I) \subseteq wp(while\ B\ do\ \alpha, Q)$ and using an inductive argument it is easy to verify that $wp(while\ B\ do\ \alpha, I) = wp(while\ B\ do\ \alpha, \neg B \bigcap I)$. So let $s \in I$ then either $s \in B$ or $s \in \neg B$. If $s \in \neg B$ then $[\![while\ B\ do\ \alpha]\!](s) = [\![(if\ B\ do\ \alpha)^0]\!](s) = null(s) = \{(s)\}$. Hence every exseq in this extree terminates cleanly and in I, so $s \in wp(while\ B\ do\ \alpha, I)$. Now suppose $s \in B$. We must show that $\forall \mathbf{x} \in [\![while\ B\ do\ \alpha]\!](s)$, $\mathbf{x}$ terminates cleanly and in I. By (c) and Lemma (3.25) $\exists m \in \mathcal{N}$ such that $[\![while\ B\ do\ \alpha]\!](s) = [\![(if\ B\ do\ \alpha)^m]\!](s)$ (and $t \notin B$ for every leaf of this extree). So we need only show that every exseq $\mathbf{x}$ in $[\![(if\ B\ do\ \alpha)^m]\!](s)$ terminates cleanly and in I. But $[\![(if\ B\ do\ \alpha)^m]\!] = [\![if\ B\ do\ \alpha]\!]^m$, hence any such $\mathbf{x}$ has m nested initial subsequences $\mathbf{x}_i$, $1 \leq i \leq m$, such that $last(\mathbf{x}_i) \in B$ for $1 \leq i < m$. But then since $s\ (= first(\mathbf{x})) \in B \bigcap I$ we get from (b) that $last(\mathbf{x}_i) \in I$ for $1 \leq i \leq m$. Hence, in particular, $last(\mathbf{x}) \in I$, as required. □

References

[1] PB Andrews, [1986], *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof* , Academic Press.

[2] RC Backhouse, [1986], *Program Construction and Verification*, Prentice-Hall, Englewood Cliffs, New Jersey.

[3] A Blikle, [1987], Proving Programs by Sets of Computations, *Lecture Notes in Computer Science 288*, Springer-Verlag, New York, p 333–358.

[4] EW Dijkstra, [1975], Guarded Commands, Nondeterminacy and Formal Derivation of Programs, *Communications of the ACM* **18** (8), p 453–458.

[5] EW Dijkstra, [1976], *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey.

[6] EW Dijkstra and CS Scholten, [1990], *Predicate Calculus and Program Semantics*, Springer-Verlag, New York.

[7] G Dromey, [1989], *Program Derivation: The Development of Programs from Specifications*, Addison-Wesley, Singapore.

[8] R Goldblatt, [1989], Varieties of Complex Algebras, *Annals of Pure and Applied Logic* **44**, p 173–242.

[9] MJC Gordon, [1989], Mechanising Programming Logics in Higher Order Logic. In G Birtwistle and PA Subrahmanyam (Eds), *Current Trends in Hardware Verification and Automated Theorem Proving*, Springer-Verlag, p 388–439.

[10] MJC Gordon *et al*, [1989], *The HOL System: DESCRIPTION*, Cambridge Research Center, SRI International.

[11] G Grätzer, [1978], *Universal Algebra*, 2nd Edition, Springer-Verlag, Berlin.

[12] D Gries, [1981], *The Science of Programming*, Springer-Verlag, New York.

[13] RC Holt, [1991], Healthiness versus Realizability in Predicate Transformers, Submitted to *Journal of Formal Aspects of Computing.*

[14] CAR Hoare, [1978], Some Properties of Predicate Transformers, *Journal of the ACM* **25**(3), p 461–480.

[15] C Morgan, [1990], *Programming from Specifications*, Prentice-Hall, Englewood Cliffs, New Jersey.

[16] JG Sanderson, [1980], *A Relational Theory of Computing, Lecture Notes in Computer Science 82*, Springer-Verlag, New York.

[17] G Grätzer and S Whitney, [1984], Infinitary Varieties of Structures Closed under the Formation of Complex Structures, *Colloq. Math* **48**, p 1–5.

[18] G Schmidt and T Ströhlein, [1985], Relation Algebras: Concepts of Point and Representability, *Discrete Mathematics* **54**, p 83–92.